Drag and Drop in Internet Explorer

The dataTransfer Object

In addition to the events that IE 5.0 introduced, the `dataTransfer` object also made its first appearance. The `dataTransfer` object is relatively simple and exists solely to help drag and drop operations transfer string data from the dragged object to the drop target.

setData() and getData()

When the `ondragstart` event fires, the `dataTransfer` object leaps into existence as a property of `window.event` (it does not exist for any events other than those listed in this article). At this point, we can use the `setData()` method to store one of two types of data: plain text or a URL. The syntax for each is as follows:

```
window.event.dataTransfer.setData("text","some text");
window.event.dataTransfer.setData("URL", "some URL");
```

It is important to note that the `dataTransfer` object will hold only one string value, so each subsequent call to `setData()` will erase the data that was previously stored in it.

After storing the string data, we can retrieve it by using the `getData()` method. Once again, we must provide the data type when doing this operation. The syntax for each case is as follows:

```
var sText = window.event.dataTransfer.getData("text");
var sURL = window.event.dataTransfer.getData("URL");
```

Your last chance to make the call to `getData()` is on the `ondrop` event. After that point, the `dataTransfer` object is destroyed along with any data that it carries.

Developer News

SCO E-Mail a Smoking Gun?

Open Source Exchange Teams With Red Hat, Novell

IDC: Web Services Consumption to Hit Stride

www Search Drag & Drop 7/15/05 IT

You may be asking yourself: What is the difference between the URL and text data types? At first glance, nothing; which is why I had to dig a little deeper to get the answer. Though the Microsoft documentation seems to overlook this, when you specify the URL data type, the user is then able to drag the object into another browser window and have it redirect to the URL that was stored. In effect, it becomes equivalent to dragging any other link into a browser window.

Take a look at this MSDN example. Here, you drag an image onto a span, and the span shows the URL of the image. Open up a second browser window and drag the image onto it. The browser will redirect to the URL of the image.

Note that in this example, the `<span>` is incorrectly indicated as a drop target. It doesn't cancel either the `ondragenter` or `ondragover` events, which is why the cursor does not change when the image is dragged over it. In short, you would never want to use the code from this example, but it does illustrate how `getData()` with a URL data type actually works.

Another interesting thing about the `dataTransfer` object is that it automatically gets loaded with text when dragging a text selection, so you can just use `getData()` when it is dropped without ever using `setData()`. Here is our earlier example with this functionality added.

dropEffect and effectAllowed

Another important use of the `dataTransfer` object is to specify what actions can be done with the dragged object and the drop target; this is where the `dropEffect` and `effectAllowed` properties come in.

The `effectAllowed` property is set *on the dragged object* and indicates which operations are allowed when the object is dropped. The possible values are uninitialized (default), none (no dropping allowed), copy, link, move, copyLink, copyMove, linkMove, and all (all effects are allowed). This property must be set during the ondragstart event:

```
window.event.dataTransfer.effectAllowed = "move";
```

If the `dataTransfer` object is storing a URL, then the `effectAllowed` property specifies which cursor is displayed when the URL is dropped onto another browser window as well as the cursor that is displayed on a drop target in the same page.

The `dropEffect` property is set *on the drop target*, indicating what the allowed drop behavior can be. This property has four possible values: none (default), move, link, and copy. Each of these values causes a different cursor to be displayed when an object is dragged over the drop target. If you want to use anything other than "copy", then `ondrop` must have its default behavior cancelled in addition to `ondragover` and `ondragenter`. This property should be set on the `ondragenter` event:

```
window.event.dataTransfer.dropEffect = "move";
```

If the values for `effectAllowed` and `dropEffect` are not compatible, then a "no-drop" cursor is shown when the object is dragged over the drop target. Compatible values contain the same word, for example, if `dropEffect` is "move" the compatible `effectAllowed` values are "move," "linkMove," and "copyMove." When `effectAllowed` is "all," it is compatible with any `dropEffect` value.

For more information on each of these properties, visit MSDN at http://msdn.microsoft.com/workshop/author/dhtml/reference/properties/effectallowed.asp and http://msdn.microsoft.com/workshop/author/dhtml/reference/properties/dropEffect.asp.

Created: October 23, 2002
Revised: October 23, 2002

Drag and Drop in Internet Explorer

Developer News

SCO E-Mail a Smoking Gun?

Open Source Exchange Teams With Red Hat, Novell

IDC: Web Services Consumption to Hit Stride

The dragDrop() Method

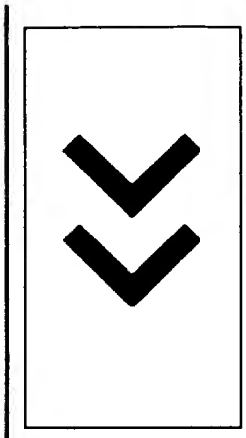
In IE 5.5 for Windows only (sorry Mac and Unix), the `dragDrop()` method was added as a method on nearly all HTML elements. This method is the only one that has to do with drag and drop behavior. While I believe that this is possibly the most misnamed method in our repertoire, it is very useful.

This method makes any element act like an image being dragged, which means that the `ondragstart`, `ondrag`, and `ondragend` events are fired in their appropriate order. (These events *will not* fire unless `dragDrop()` is called .) The problem, of course, is when to call the `dragDrop()` method in order to accurately mimic dragging an image or text. This is what we will explore now.

When you think about dragging something on the screen, you think about moving the cursor over it, pressing the mouse button down, and then moving the mouse again. The dragging doesn't actually start until the cursor is moved, which is our hint: we can use the `onmousemove` event to begin a drag.

The `onmousemove` event fires continuously as the cursor moves over the element, which obviously isn't what we want. We care only if the cursor moves *and* the left mouse button is down. We can check the status of the mouse button by using `event.button` and checking to see if it's equal to 1 (which indicates that the left button is down). So, our `onmousemove` event handler should look like this (note that `DraggableElement` should be replaced by a reference to a real element):

```
function handleMouseMove() {
  if (window.event.button == 1) {
    DraggableElement.dragDrop();
```




```
    }
}
```

The other necessary step is to add an `ondragstart` event handler that sets data to the `dataTransfer` object (otherwise, there is no data to get when it's dropped). This function would look something like this:

```
function handleDragStart() {
    window.event.dataTransfer.setData("text", "Text to send")
}
```

Employing these methods, I have constructed another example. This one has a textbox to drag text from as well as a blue DIV that can be dragged as well. Just for fun, I also added some color change for the target DIV when something is dragged over it.

Conclusion

IE's built-in drag and drop functionality is very powerful, if not cool. It offers a wide range of possibilities for Web developers, whether they be working on Web sites or Web applications. When you consider that things can be dragged across frames, and even into other browser windows, developers can improve the usability of Web interfaces dramatically. The only real downside is the techniques discuessed here only work in IE, so before throwing this into your project, be sure to plan strategies for other browsers.

About the Author

Nicholas C. Zakas is a user interface designer for Web applications at MatrixOne, Inc. in Massachusetts. Nicholas works primarily as a client-side developer using JavaScript, DHTML, XML and XSLT. He can be reached via e-mail at nicholas@nczonline.net or at his Web site, http://www.nczonline.net.

home / programming / javascript / dragdropie

[previous]

Drag & Drop

Use

You use Drag & Drop to do the following:

- Open objects

  You can call the object editor for this object by dragging the object from the navigation area to the work area. To do this, drag the object to the edge of the work area. When this edge is selected and turns black, drop the object.

  This is supported within the Integration Repository and the Integration Directory.

- Create object references

  The object editors of some object types contain Drag & Drop targets. These drop targets are indicated by the hand icon. If object O1 is opened in change mode and object O2 is dragged from the navigation area to the hand icon using Drag & Drop, then object O2 is entered at this location in the object editor of object O1. This creates an object reference from object O1 to object O2.

  If you have opened a message interface in change mode, then you can drag a message type from the navigation area to the *Output Message Type* line of the message interface using Drag & Drop. To do this, drag the message type to the hand icon in the object editor of the message interface. The message type is entered in the *Output Message Type* line and an object reference is created to the message type.

  Drag & Drop is also supported between the Integration Repository and the Integration Directory.

  You can use Drag & Drop to drag an interface object from the navigation area of the Integration Repository to the hand icon in the *Inbound Interface* column in the *Edit Interface Determination* editor in the Integration Directory. This creates an object reference from the interface determination in the Integration Directory to a message interface in the Integration Repository.

VOODOO'S INTRODUCTION TO JAVASCRIPT

Part 12: Drag & Drop

What is drag & drop?

With the help of the new event model of JavaScript 1.2 and layers we can implement drag & drop on our web-page. You'll need **at least Netscape Navigator 4.0 for this as we use JavaScript 1.2 features**.

What is drag & drop? Some operating systems (like Win95/NT or MacOS) let you for example erase files through dropping icons on a trash bin. What you do is you click on the icon of a file, drag (i.e. you hold the mouse button down while moving the mouse) the icon to the trash bin and drop it there.
The drag & drop we want to implement here **is restricted to the web-page**. So you cannot use this code shown here in order to drag objects inside a HTML-page to your hard disk or something like this. (Since Netscape Navigator 4.0 your script can react to an event called *DragDrop* when somebody drops a file on your browser window - but this is *not* what we are going to talk about in this lesson)

You can now test the example we are going to develop in this lesson. After you pushed this button click on the different objects and move them around:

Drag & Drop example

You might also want to check out the example provided by Netscape. You can find it at this address:
http://home.netscape.com/comprod/products/communicator/user_agent_vacation.html

JavaScript does not support drag & drop directly. This means we cannot just specify a property *dragable* (or whatever) in an image object. We have to write the code for this on our own. You'll see that this isn't too difficult.
So what do we need? We need two things. First we have to register certain mouse events, i.e. how do we know which object shall be moved to which position? Then we need to make up our minds on how we can display the moving objects on the screen. Of course we will use the new layer feature for defining different objects and moving them around on the screen. Every object is represented through its own layer.

Mouse events with JavaScript 1.2

Which mouse events do we have to use? We don't have a *MouseDrag* event - but we can achieve the same through the events *MouseDown*, *MouseMove* and *MouseUp*. JavaScript 1.2 uses a new event model. Without this event model we could not solve our task. I have talked about the new event model in the last lesson. But let's have a look at the important parts once again.
The user pushes the mouse button somewhere inside the browser window. Our script has to react on this event and calculate which object (i.e. layer) was hit. We need to know the coordinates of the mouse event. JavaScript 1.2 implements a new Event object which stores the coordinates of a mouse event (besides other information).
Another important thing is called event capturing. If a user for example clicks on a button the mouse

event is sent directly to the button object. But in our case we want the window to handle our event. So we let the window *capture* the mouse event, i.e. that the window object gets this event and can react upon it. The following example demonstrates this (using the event *Click*). You can click somewhere inside the browser window. An alert window pops up and displays the coordinates of the mouse event:

Demonstration of Click

This is the code for this example:

```
<html>

<script language="JavaScript">
<!--

  window.captureEvents(Event.CLICK);

  window.onclick= displayCoords;


  function displayCoords(e) {
    alert("x: " + e.pageX + " y: " + e.pageY);
  }

// -->
</script>

Click somewhere inside the browser window.

</html>
```

First we tell the window object to capture the *Click* event. We use the method *captureEvent()* for this. The line

```
  window.onclick= displayCoords;
```

defines what happens when a *Click* event ocurs. It tells the browser to call *displayCoords()* as a reaction to a *Click* event (Please note that you *must not* use brackets behind *displayCoords* in this case). *displayCoords()* is a function which is defines like this:

```
  function displayCoords(e) {
    alert("x: " + e.pageX + " y: " + e.pageY);
  }
```

You can see that this function takes one argument (we call it *e*). This is the Event object which is being passed to the *displayCoords()* function. The Event object has got the properties *pageX* and *pageY* (besides others) which represent the coordinates of the mouse event. The alert window displays these values.

MouseDown, MouseMove and MouseUp

As I already told you JavaScript does not know a MouseDrag event. Therefore we have to use the events *MouseDown*, *MouseMove* and *MouseUp* in order to implement drag & drop. The following example demonstrates the use of *MouseMove*. The actual coordinates of the mouse cursor are displayed on the

statusbar.

Demonstration of MouseMove

You can see that the code is almost the same as in the last example:

```
<html>

<script language="JavaScript">
<!--

  window.captureEvents(Event.MOUSEMOVE);

  window.onmousemove= displayCoords;


  function displayCoords(e) {
    status= "x: " + e.pageX + " y: " + e.pageY;
  }

// -->
</script>

Mouse coordinates are displayed on the statusbar.

</html>
```

Please note that you have to write *Event.MOUSEMOVE*, where *MOUSEMOVE* must be in upper case. When defining which function to call when the MouseMove event occurs you have to use lower case: *window.onmousemove=...*

Now we can combine the last two examples. We want the coordinates of the mouse pointer to be displayed when the mouse is being *moved with pushed mouse button*. The following example demonstrates this:

Test 'MouseDrag'

The code for this example looks like this:

```
<html>

<script language="JavaScript">
<!--

window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);

window.onmousedown= startDrag;
window.onmouseup= endDrag;
window.onmousemove= moveIt;

function startDrag(e) {
  window.captureEvents(Event.MOUSEMOVE);
}

function moveIt(e) {
```

```
  // display coordinates
  status= "x: " + e.pageX + " y: " + e.pageY;
}

function endDrag(e) {
  window.releaseEvents(Event.MOUSEMOVE);
}


// -->
</script>

Push the mouse button and move the mouse. The coordinates are being
displayed on the statusbar.

</html>
```

First we tell the window object to capture the events *MouseDown* and *MouseUp*:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);
```

You can see that we use the sign | (*or*) in order to define several events which shall be captured by the window object. The next two lines define what happens when these events occur:

```
window.onmousedown= startDrag;
window.onmouseup= endDrag;
```

The next line of code defines what happens when the window object gets a *MouseMove* event:

```
window.onmousemove= moveIt;
```

But wait, we didn't define *Event.MOUSEMOVE* in *window.captureEvents()*! This means that this event isn't captured by the window object. So why do we tell the window object to call *moveIt()* although this event never reaches the window object? The answer to this question can be found in the function *startDrag()* which is being called as soon as a *MouseDown* event occurs:

```
function startDrag(e) {
  window.captureEvents(Event.MOUSEMOVE);
}
```

This means the window object captures the *MouseMove* event as soon as the mouse button is pushed down. We have to stop capturing the *MouseMove* event when the *MouseUp* event occurs. This does the function *endDrag()* with the help of the method *releaseEvents()*:

```
function endDrag(e) {
  window.releaseEvents(Event.MOUSEMOVE);
}
```

The function *moveIt()* writes the mouse coordinates to the statusbar.

Now we have all elements for registering the events needed to implement drag & drop. We can move forward to displaying the objects on the screen.

Displaying moving objects

We have seen in previous lessons that we can create moving objects with the help of layers. All we have to do now is to register which object the user clicked on. Then this object has to follow the mouse movements. Here is the code for the example shown at the beginning of this lesson:

```
<html>
<head>

<script language="JavaScript">
<!--

var dragObj= new Array();
var dx, dy;

window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);

window.onmousedown= startDrag;
window.onmouseup= endDrag;
window.onmousemove= moveIt;

function startDrag(e) {
  currentObj= whichObj(e);
  window.captureEvents(Event.MOUSEMOVE);
}

function moveIt(e) {
  if (currentObj != null) {
    dragObj[currentObj].left= e.pageX - dx;
    dragObj[currentObj].top= e.pageY - dy;
  }
}

function endDrag(e) {
  currentObj= null;
  window.releaseEvents(Event.MOUSEMOVE);
}

function init() {
  // define the 'dragable' layers
  dragObj[0]= document.layers["layer0"];
  dragObj[1]= document.layers["layer1"];
  dragObj[2]= document.layers["layer2"];
}

function whichObj(e) {

  // check which object has been hit

  var hit= null;
  for (var i= 0; i < dragObj.length; i++) {
    if ((dragObj[i].left < e.pageX) &&
        (dragObj[i].left + dragObj[i].clip.width > e.pageX) &&
        (dragObj[i].top < e.pageY) &&
        (dragObj[i].top + dragObj[i].clip.height > e.pageY)) {
          hit= i;
          dx= e.pageX- dragObj[i].left;
          dy= e.pageY- dragObj[i].top;
          break;
    }
  }
```

```
  return hit;
}


// -->
</script>
</head>
<body onLoad="init()">

<layer name="layer0" left=100 top=200 clip="100,100" bgcolor="#0000ff">
<font size=+1>Object 0</font>
</layer>

<layer name="layer1" left=300 top=200 clip="100,100" bgcolor="#00ff00">
<font size=+1>Object 1</font>
</layer>

<layer name="layer2" left=500 top=200 clip="100,100" bgcolor="#ff0000">
<font size=+1>Object 2</font>
</layer>

</body>
</html>
```

You can see that we define three layers in the <body> part of this HTML-page. After the whole page is loaded the function *init()* is called through the *onLoad* event handler in the <body> tag:

```
function init() {
  // define the 'dragable' layers
  dragObj[0]= document.layers["layer0"];
  dragObj[1]= document.layers["layer1"];
  dragObj[2]= document.layers["layer2"];
}
```

The *dragObj* array takes all layers which can be moved by the user. Every layer gets a number in the *dragObj* array. We will need this number later on.
You can see that we use the same code as shown above in order to capture the mouse events:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);

window.onmousedown= startDrag;
window.onmouseup= endDrag;
window.onmousemove= moveIt;
```

I have added the following line to the *startDrag()* function:

```
currentObj= whichObj(e);
```

The function *whichObj()* determines which object the user clicked on. It returns the number of the layer. If no layer has been hit it returns the value *null*. The variable *currentObj* stores this value. This means that *currentObj* represents the number of the layer which is being moved at the moment (or it is *null* if no layer is being moved).

In the function *whichObj()* we check the properties *left*, *top*, *width* and *height* for each layer. With the help of these values we can check which object the user clicked on.

Dropping objects

We have now everything we need in order to implement drag & drop. With our script the user can drag around objects on our web-page. But we haven't talked about dropping objects yet. Let's suppose that you want to create an online shop. You have several items which can be put into a shopping basket. The user has to drag these items to the shopping basket and drop them there. This means we have to register when the user drops an object on the shopping basket - which means that he wants to buy it.
Which part of the code do we have to change in order to implement this? We have to check which position the object has after a *MouseUp* event - i.e. we have to add some code to the function *endDrag()*. We could for example check if the coordinates of the mouse event lie inside a certain rectangle. If this is true you call a function which registers all items to buy (you might want to put them inside an array). Then you could show the item inside the shopping basket.

Improvements

There are several ways for improving our script. First we might want to change the order of the layers as soon as the user clicks on one object. Otherwise it might a look a bit strange if you move an object and it disappears behind another object. You can solve this problem by changing the order of the layers in the *startDrag()* function.
I don't think that you'll be satisfied by putting up red, green and blue boxes on your web page. Add some cool graphics and the users will remember your page. You can place anything inside the layer objects. So place a single <img> tag there if you want your object to appear as an image.

[previous] [index]

e-mail:skoch@rumms.uni-mannheim.de
http://rummelplatz.uni-mannheim.de/~skoch/
My JavaScript-book

[Next][Previous][Up]

Advanced Features

Motif Drag and Drop

EditTable supports full Motif drag and drop functionality (Note: drag and drop is not supported under X11R4/Motif1.1). One can drag and drop cells from a table into another table, from a text widget into a table, etc. If you are using ChartObject, it is also possible to drag and drop data from a table to a chart and from a chart to a table.

EditObject action *MotifStartDrag* controls the drag operation, and connects with the following translation:

```
<Btn2Down>: MotifStartDrag()
```

Drag and Drop operations can be disabled by setting EditObject resources **XmNallowDrag** and **XmNallowDrop** to False. Also, callback *XmNdragDropCallback* is invoked on both drag and drop operations and can be used to selectively enable or disable either drag or drop operations. This callback can also be used to modify the type of drag and drop operation (copy/move/link) and modify the index or count of the cells being dragged.

Default behavior for drag and drop

The following table summarizes the default behavior for drag and drop. if there is a key sequence in the translation, it is important that the keys remain pressed until the drag and drop operation has been completed and the button has been released.

Key Sequence	Drag Source	Drop Site	Description
<Btn2Down>	EditTable	EditTable	Copies the selected cells from the source table to the destination table.
<Btn2Down>	EditTable	Chart object	The data contained in the selected cells is copied into the chart object (move option is disabled for the EditTable widget).
Ctrl <Btn2Down>	EditTable	Chart object	The data contained in the selected cells is copied into the chart object.
Ctrl Shift <Btn2Down>	EditTable	Chart object	The data contained in the selected cells is copied into the chart object and a link is made between the table and the chart.
<Btn2Down>	Chart object	EditTable	The data contained in the chart is moved to the table at the location specified by the pointer.
Ctrl	Chart object	EditTable	The data contained in the chart is copied to the table at the location specified by the pointer.

<Btn2Down>			
Ctrl Shift <Btn2Down>	Chart object	EditTable	The data contained in the chart is copied to the table at the location specified by the pointer. A link is established between the chart and the table.
<Btn2Down>	Motif Text	EditTable	Content of the Text widget is copied into the specified table cell.

Default behavior for drag and drop

[Next][Previous][Up]

Drag and drop overview

Classes: wxDataObject, wxTextDataObject, wxDropSource, wxDropTarget, wxTextDropTarget, wxFileDropTarget

Note that wxUSE_DRAG_AND_DROP must be defined in setup.h in order to use drag and drop in wxWidgets.

See also: wxDataObject overview and DnD sample

It may be noted that data transfer to and from the clipboard is quite similar to data transfer with drag and drop and the code to implement these two types is almost the same. In particular, both data transfer mechanisms store data in some kind of wxDataObject and identify its format(s) using the wxDataFormat class.

To be a *drag source*, i.e. to provide the data which may be dragged by the user elsewhere, you should implement the following steps:

- **Preparation:** First of all, a data object must be created and initialized with the data you wish to drag. For example:

```
wxTextDataObject my_data("This text will be dragged.");
```

- **Drag start:** To start the dragging process (typically in response to a mouse click) you must call wxDropSource::DoDragDrop like this:

```
wxDropSource dragSource( this );
dragSource.SetData( my_data );
wxDragResult result = dragSource.DoDragDrop( TRUE );
```

- **Dragging:** The call to DoDragDrop() blocks the program until the user releases the mouse button (unless you override the GiveFeedback function to do something special). When the mouse moves in a window of a program which understands the same drag-and-drop protocol (any program under Windows or any program supporting the XDnD protocol under X Windows), the corresponding wxDropTarget methods are called - see below.

- **Processing the result:** DoDragDrop() returns an *effect code* which is one of the values of `wxDragResult` enum (explained here):

```
switch (result)
{
    case wxDragCopy: /* copy the data */ break;
    case wxDragMove: /* move the data */ break;
    default:         /* do nothing */ break;
}
```

To be a *drop target*, i.e. to receive the data dropped by the user you should follow the instructions below:

- **Initialization:** For a window to be a drop target, it needs to have an associated wxDropTarget object. Normally, you will call wxWindow::SetDropTarget during window creation associating your drop target with it. You must derive a class from wxDropTarget and override its pure virtual methods. Alternatively, you may derive from wxTextDropTarget or wxFileDropTarget and override their OnDropText() or OnDropFiles() method.
- **Drop:** When the user releases the mouse over a window, wxWidgets asks the associated wxDropTarget object if it accepts the data. For this, a wxDataObject must be associated with the drop target and this data object will be responsible for the format negotiation between the drag source and the drop target. If all goes well, then OnData will get called and the wxDataObject belonging to the drop target can get filled with data.
- **The end:** After processing the data, DoDragDrop() returns either wxDragCopy or wxDragMove depending on the state of the keys <Ctrl>, <Shift> and <Alt> at the moment of the drop. There is currently no way for the drop target to change this return code.

Implementing Drag and Drop Functionality - by Borland Developer Sup


```
 Technical Information Database

TI1482D.txt   Implementing Drag and Drop Functionality
Category   :Application Interop
Platform    :All
Product    :Delphi  All

Description:
This document describes the process of creating a drag and drop
functionality that could be applied to text items and graphical
objects. Regular text indicates the procedural steps to execute
to achieve the functionality. Text in the parentheses is
explanation and background information.

1. Place the two edit boxes onto your form. {One will serve as
the source for the text that you will drag and drop. The other
will serve as the destination you will drop the text onto.}

2. Select the first edit box and name it SourceEdit by
entering that name in the name property in the Object
Inspector.

3. Select the second edit box and name it SenderEdit. {The
names source and sender are not necessary. They are
suggested because "source" and "sender" are default variables
given to work with in the OnDragOver and OnDragDrop event
procedure code blocks. The source variable is where the
drag operation began, and the sender variable is the control
that was dropped onto. See the help topics related to
messages for further information on this and related topics.}

4. Click the SourceEdit component so it is selected and go to
the Object Inspector. Set its DragMode property to
dmAutomatic.

5.  Select the SenderEdit and go to the events page of the
Object Inspector and double click on the OnDragOver  event.

6. Type exactly what is in the quotes here where the cursor
is between the begin and end statements: "Accept := True;".
{Notice that the Accept variable is supplied for you by
default inside of the OnDragOver event procedure code block.}
```

7. Now go to the events page of the Object Inspector and double click on the OnDragDrop event.

8. Type exactly what is in the quotes here where the cursor is between the begin and end statements:
" SenderEdit.Text := SourceEdit.Text ".

Now if you run the application you will be able to click down on the SourceEdit component and drag and drop onto the SenderEdit component. When you drop onto the SenderEdit it changes its text to whatever was in the SourceEdit.

Reference:

7/16/98 4:34:13 PM

OLE Drag and Drop

Part 6 - Drop Targets

Download full source and demo (4Kb)

Welcome to the sixth part of the "OLE Drag and Drop" tutorial series! This article will concentrate on implementing a small application which will act as a drop-target. What this means is that our application will be capable of receiving objects (be they files, pictures or text) which are dragged onto it.

We will implement an IDropTarget COM interface which will allow any OLE application to drag it's data over our application. This will take the form of a simple EDIT control which can act as a target for dropped CF_TEXT data. Hopefully you will be able to take the code presented here and "drag" it straight into your own apps ;-)

Become a "Drop Target"

In order for a window to accept data from a drap-drop operation, it must be registered as a "drop target". There is an OLE API call - **RegisterDragDrop** - which is used for this very purpose. The function prototype looks like this:

```
WINOLEAPI RegisterDragDrop(HWND hwnd, IDropTarget * pDropTarget);
```

The first parameter to this function is the window handle, of the window that is destined to be a drop target. The second parameter is a pointer to the IDropTarget COM object. The COM/OLE runtime will call the methods on this interface during the course of a drag-drop operation.

Likewise there is an OLE API call to remove drag-and-drop functionality from a window:

```
WINOLEAPI RevokeDragDrop(HWND hwnd);
```

All that is required from us is to called RegisterDragDrop when our window is created, and RevokeDragDrop when our window is destroyed. Before we can call RegisterDragDrop though, we need to construct a COM object which supports the IDropTarget interface.

The IDropTarget Interface

The IDropTarget Interface is relatively simple, with only four functions that need to be implemented. Of course there is also the IUnknown interface which needs to be implemented but we have already covered that earlier.

IDropTarget Methods	Description
DragEnter	Determines whether a drop can be accepted and its effect if it is accepted.
DragOver	Provides target feedback to the user through the **DoDragDrop** function.
DragLeave	Causes the drop target to suspend its feedback actions.
Drop	Drops the data into the target window.

Each one of these four functions will be called by the COM/OLE runtime whenever an "object" is dragged over our registered window. Each function has a different task, as shown in the table above. It is up to us to provide the implementations of these functions.

Implementing IDropTarget

The IDropTarget interface is (in my experience) very difficult to write without using "application specific" code. i.e. there is no easy way to make a generic IDropTarget COM object which can be re-used between all of your applications.

This is because the requirement of IDropTarget is to show graphical feedback in your target window whenever an object is dragged over it, and also the application-specific code to access the data object's content.

Out of all the drag+drop interfaces, the IDropTarget is the one that would be best integrated directly into your window class. For example, supposing you have implemented a custom window using a C++ class - the best method to add drop-target support to this window is have your custom-window class inherit directly from IDropTarget, rather than having a separate CDropTarget class. This means that your drop-target code would have full access to all of your internal window state.

However, for the time-being here is the CDropTarget class in all it's glory:

```
class CDropTarget : public IDropTarget
{
public:
    // IUnknown implementation
    HRESULT __stdcall QueryInterface (REFIID iid, void ** ppvObject);
    ULONG   __stdcall AddRef (void);
    ULONG   __stdcall Release (void);

    // IDropTarget implementation
    HRESULT __stdcall DragEnter(IDataObject * pDataObject, DWORD grfKeyState, P
    HRESULT __stdcall DragOver(DWORD grfKeyState, POINTL pt, DWORD * pdwEffect)
```

```
    HRESULT __stdcall DragLeave(void);
    HRESULT __stdcall Drop(IDataObject * pDataObject, DWORD grfKeyState, POINTL

    // Constructor
    CDropTarget(HWND hwnd);
    ~CDropTarget();


private:

    // internal helper function
    DWORD DropEffect(DWORD grfKeyState, POINTL pt, DWORD dwAllowed);
    bool  QueryDataObject(IDataObject *pDataObject);

    // Private member variables
    long   m_lRefCount;
    HWND   m_hWnd;
    bool   m_fAllowDrop;

    // Other internal window members

};
```

As well as the reference count, we need to store two additional variables: The **m_hWnd** variable is the window handle of the drop-target is needed so we can provide visual feedback during the drag-drop operation. The **m_fAllowDrop** is used to indicate whether or not the dataobject being dropped on us contains useful data. This is so we don't have to continually query the dataobject - basically its an optimization trick.

IDropTarget::DragEnter

Let's look at the IDropTarget::DragEnter function first of all, because this is the first function that is called by COM when an object is dragged over our window:

```
HRESULT DragEnter(
   IDataObject * pDataObject,    // Pointer to the interface of the source data
   DWORD         grfKeyState,    // Current state of keyboard modifier keys
   POINTL        pt,             // Current cursor coordinates
   DWORD *       pdwEffect       // Pointer to the effect of the drag-and-drop
   );
```

Look closely at the function prototype above, because it is important to understand what each of the parameters are used for.

- **IDataObject** - the very first argument is another pointer to the data object passed to us (via COM) by the source of the drag-drop operation. The IDataObject is simply the "transport medium" for the data that is being dropped. We can query the data object during DragEnter to see if it has any data that we want.

- **gfrKeyState** - holds the state of the keyboard modifier keys such as Control, Alt and Shift, and the state of the mouse buttons. It's a simple DWORD variable comprised using one or more of the following bit-flags: MK_CONTROL, MK_SHIFT, MK_ALT, MK_BUTTON, MK_LBUTTON etc.

- **pt** - a POINTL structure, containing the coordinates of the mouse as it enters our window. In some applications this parameter would be used to check if the mouse was positioned over allowable drop areas, or used simply to position some kind of "insertion" cursor to indicate where the dropped data would go.

- **pdwEffect** - pointer to a DWORD value that specifies the drop-effects that are *allowed* by the drop-source. This value is the same as the **dwOKEffect** value specified by the caller of **DoDragDrop**.

Our implementation of DragEnter needs to perform several common tasks, in addition to drawing graphical feedback.

1. Inspect the supplied data object and decide if it contains any useful data or not.
2. Inspect the keyboard state stored in **grfKeyState** and calculate what the drop-effect should be. i.e. if the Control key is held down, the drop-effect should be "copy", if Shift is held down, the drop-effect should be "move".
3. Verify that the computed drop-effect is compatible with those allowed by the drop-source.
4. Store the final drop-effect in the DWORD pointed to by **pdwEffect**.

Don't get caught up in the complexity of all this. The purpose of DragEnter is to simply say "yes or no" to the drag-drop operation, and to specify what the drop-effect should be so that the mouse-cursor can be updated by OLE.

```
HRESULT __stdcall CDropTarget::DragEnter(IDataObject *pDataObject, DWORD grfKey
                                         POINTL pt, DWORD *pdwEffect)
{
    // does the dataobject contain data we want?
    m_fAllowDrop = QueryDataObject(grfKeyState, pdwEffect, pDataObject);

    if(m_fAllowDrop)
    {
        // get the dropeffect based on keyboard state
        *pdwEffect = DropEffect(grfKeyState, pt, *pdwEffect);

        SetFocus(m_hWnd);

        PositionCursor(m_hWnd, pt);
    }
    else
    {
        *pdwEffect = DROPEFFECT_NONE;
    }
```

```
    return S_OK;
}
```

Apart from setting focus to the underlying window and positioning the EDIT caret on the nearest character under the mouse, the DragEnter function has been simplified by delegating the functionality to two internal helper routines:

```
bool CDropTarget::QueryDataObject(IDataObject *pDataObject)
{
    FORMATETC fmtetc = { CF_TEXT, 0, DVASPECT_CONTENT, -1, TYMED_HGLOBAL };

    // does the data object support CF_TEXT using a HGLOBAL?
    return pDataObject->QueryGetData(&fmtetc) == S_OK ? true : false;
}
```

QueryDataObject is a private member function which is used purely to inspect the supplied data object, and decide if it contains data that is meaningful to our drop-target. In our case, we only accept CF_TEXT data stored as a HGLOBAL, so this is what we ask for. A private member variable **m_fAllowDrop** is used to remember this decision.

```
DWORD CDropTarget::DropEffect(DWORD grfKeyState, POINTL pt, DWORD dwAllowed)
{
        DWORD dwEffect = 0;

        // 1. check "pt" -> do we allow a drop at the specified coordinates?

        // 2. work out that the drop-effect should be based on grfKeyState
        if(grfKeyState & MK_CONTROL)
        {
                dwEffect = dwAllowed & DROPEFFECT_COPY;
        }
        else if(grfKeyState & MK_SHIFT)
        {
                dwEffect = dwAllowed & DROPEFFECT_MOVE;
        }

        // 3. no key-modifiers were specified (or drop effect not allowed), so
        //    base the effect on those allowed by the dropsource
        if(dwEffect == 0)
        {
                if(dwAllowed & DROPEFFECT_COPY) dwEffect = DROPEFFECT_COPY;
                if(dwAllowed & DROPEFFECT_MOVE) dwEffect = DROPEFFECT_MOVE;
        }

        return dwEffect;
}
```

The **DropEffect** helper function is used to compute the drop-effect based on the keyboard state and the effects allowed by the source.

First of all the **grfKeyState** variable is checked to see if either the Control or Shift keys

are being used. The standard OLE behaviours for these keys are that Control should force a *Copy* of data, and Shift should force a *Move* of data. If both are held down, the the data should be *Linked* (i.e. the source should make a shortcut to the target), but we don't support this feature.

The important thing to note is the use of the "bitwise-AND" operator when assigning the drop-effect to dwEffect:

```
dwEffect = dwAllowed & DROPEFFECT_COPY;
```

The result of this assignment is simple - **dwEffect** will be assigned the value DROPEFFECT_COPY, but only if this value is present in the **dwAllowed** variable. This use of logic prevents us from forcing a drop-effect that is not allowed by the source.

The next stage is to decide what to do if no keyboard modifiers are present - i.e. Control or Shift are not in use. In this case we simply inspect the drop-effects allowed by the source and choose (in order of priority) which one to use - in our implementation, we let **data moves** override **data copies**.

IDropTarget::DragOver

The DragOver function will be called multiple times during the lifetime of a drag-drop operation. Therefore it is important for this function to be efficiently written. DragOver is called whenever the state of the keyboard modifiers change (i.e. shift/control etc), or when the mouse moves. It is the responsibilty of this function to indicate to OLE what the drop-effect will be based on the state of the keyboard and mouse position.

```
HRESULT __stdcall CDropTarget::DragOver(DWORD grfKeyState, POINTL pt, DWORD * p
{
    if(m_fAllowDrop)
    {
        *pdwEffect = DropEffect(grfKeyState, pt, *pdwEffect);
        PositionCursor(m_hWnd, pt);
    }
    else
    {
        *pdwEffect = DROPEFFECT_NONE;
    }

    return S_OK;
}
```

DragOver is very simple to write, because the logic is identical to that of DragEnter. We use the previously computed **m_fAllowDrop** and the **DropEffect** helper routine to return a drop-effect through the **pdwEffect** pointer.

IDropTarget::DragLeave

The DragLeave function is called whenever the mouse cursor is moved outside of our drop-target window, or the Escape key is pressed which cancels the drag-drop operation. It's prototype (and implementation) is really simple:

```
HRESULT __stdcall CDropTarget::DragLeave(void)
{
    return S_OK;
}
```

This is the most basic way to write this function. The only reason this function exists is so that applications that make heavy use of graphical feedback effects get a chance to clean up once the mouse moves out of the window. For example, imagine the following scenario: whenever something is dragged over a drop-target, the DragEnter function is used to change the colour of the window-border. In this case, the DragLeave function would be used to restore the window-border.

IDropTarget::Drop

The Drop function's prototype is exactly the same as the DragEnter function:

```
HRESULT __stdcall CDropTarget::Drop(IDataObject *pDataObject, DWORD grfKeyState
{
    PositionCursor(m_hWnd, pt);

    if(m_fAllowDrop)
    {
        DropData(m_hWnd, pDataObject);

        *pdwEffect = DropEffect(grfKeyState, pt, *pdwEffect);
    }
    else
    {
        *pdwEffect = DROPEFFECT_NONE;
    }

    return S_OK;
}
```

This function is called when OLE has determined that the drag-drop will go ahead. We get the same interface pointer to the IDataObject that we received during DragEnter, which we can now retrieve data from to paste into our edit window.

The **DropData** helper function is used to access the CF_TEXT data inside the dataobject and insert it into the edit control. This routine is purely academic and as we already know how to access a dataobject I won't bother detailing it any further - just look at the sourcecode download if you are interested.

Conclusion

We've done it! It's taken six tutorials to get to this stage, but it was necessary to break up the subject matter into managable chunks.

So what have we accomplished?

At this stage we know how to implement IDataObject, IEnumFORMATETC, IDropTarget and IDropSource, as well as access the Windows clipboard using the new OLE functions.

There is still scope for further tutorials though. The next tutorial (or two) will look at dragging and dropping files (and filenames), and also using the IStream COM interface to stream file content between applications and the Windows Shell.

As always, I'd like to hear any feedback you may have on this tutorial series. More feedback equals more tutorials, so stay tuned!

Click here to return to the start of the series: OLE Drag and Drop

Please send any comments or suggestions to: james@catch22.net

Last modified: 16 February 2005 20:02:26